

HURDMEETING 2005

4 : 11 : 05.

Lo primero que hacemos es instalar **Qemu**. La instalación "a pelo" sobre hardware real es un poco arriesgada. **Qemu** es un programa libre de emulación de hardware, al estilo de **Vmware**. Se descarga desde qemu.org. También es necesaria una imagen, es decir un fichero virtual que contenga el sistema operativo que vayamos a emular. Se pueden descargar imágenes desde la página web de **Qemu**. Pueden encontrarse imágenes muy buenas en otros sitios web. El formato de estas suele ser **.iso**

- Charla Primera. PRESENTACION. Por Marchesi.

Este es el primer **Hurdmeeting** que tiene lugar en Madrid. También es el primero que tiene lugar en España, y en Europa. De hecho es el primer **Hurdmeeting** de la historia, en todo el mundo. Nadie lo había intentado antes porque seguramente nadie se había atrevido.

La asistencia ha superado, con creces todas las expectativas que se esperaban. De hecho temíamos una situación muy diferente. Esto es debido a que este evento tiene unas características bastante especiales. Para empezar, Es un encuentro para desarrolladores, sobre un kernel bastante peculiar. Y para continuar es un kernel que todavía no funciona correctamente. Hay quien se pregunta porque todavía continua su desarrollo después de más de diez años de trabajo.

El **Hurd** es un núcleo que presenta bastante dificultades a sus desarrolladores. El objetivo de este encuentro es formar a hackers para que más adelante puedan incorporarse al grupo de desarrollo. Así que el público en general no tiene interés por este tipo de encuentros, y la mayoría de los hackers no están preparados para trabajar en un proyecto tan complicado. Bueno, y entonces ¿Quién iba a tomarse la molestia de venir al meeting? No lo sabíamos. Ahora si lo sabemos. Más de cincuenta Hackers lo han hecho. Algunos han venido de otros países. Un grupo que finalmente no ha podido acudir, lo hacía desde Argentina. Este **Hurdmeeting 1** es una apuesta valiente para salvar un proyecto, y quizás para obtener, dentro de algún tiempo, un kernel preparado para el trabajo en producción. Para eso estamos aquí. La importancia de un logro así sería enorme. Para comprenderlo pasamos a contar la historia de **Hurd**.

El desarrollo de **Hurd** comenzó en 1989. El Sistema **Gnu** en 1984. En aquél momento el número de hackers de **Gnu** eran bastante escaso. Se desarrolló **Emacs**, el enlazador, y otras cosas como **grep**, etc. Pero era necesario un núcleo. Se buscaron varias alternativas. Primero, se intentó con un núcleo llamado **Trux**. Se tenía un gran deseo de hacer un sistema funcional para **Gnu**. Pero se cometió un gran error (reconocido por **FSF** y **GNU**). Se diseñó un microkernel muy innovador, basado en nuevas ideas sobre

arquitectura de núcleos. Han pasado muchos años desde entonces, pero **Hurd** ni se ha terminado, ni se ha abandonado.

La aspiración del proyecto **GnuHurd** era y es, escribir un núcleo que incremente las funcionalidades de los núcleos monolíticos como **Linux**. El problema fue que se intento crear el mejor núcleo de la historia, en un momento (en los comienzos del sistema **Gnu**) en el que lo que hacia falta era un núcleo. Cualquier núcleo libre que fuera funcional. La historia del kernel de **Linux** y **GNU** es bien conocida. Para solventar la situación se utilizó el núcleo de **Torbalds**. Y desde entonces no ha habido grandes cambios en este tema.

La situación actual del proyecto **Hurd** es ventajosa en parte, y también a la inversa. La desventaja principal, es que hay pocos recursos en este proyecto (debido a que ya no se considera un objetivo principal dotar a **Gnu** de un núcleo, puesto que ya tenemos Linux). La ventaja es que podemos dedicarle el tiempo que sea necesario. Desde el año 2001 el proyecto está bastante parado. La causa es la carencia de desarrolladores. En estas situaciones, la solución parece sencilla: Se buscan. Pero ya se ha intentado este método y no ha resuelto el problema.

Creemos que el problema real es la dificultad de este núcleo. La mayoría de hackers no están preparados para convertirse, sin una formación previa, en desarrolladores de **Hurd**. Cuando terminemos el meeting, estaremos preparados para comprender la problemática del **Hurd**. Así como las ventajas y desventajas de los micronúcleos. En definitiva, para tener una opinión formada sobre el tema. (en lugar de decir que los micronúcleos son lentos, como hacen algunos trolls de Barrapunto).

Lo primero que se va a hacer en el meeting, es un taller de instalación, del tipo "**Hurd from scratch**", es decir instalación desde cero, a cargo de **Pancake**.

- Charla Segunda. Hurd from scratch. Por Pancake.

Esta charla trata de aspectos básicos. Donde encontrar los sources, como instalar el **Hurd**, etc. No da tiempo a una instalación completa, ya que la compilación puede tardar varias horas. Es muy importante que el desarrollador pueda compilar **Hurd** por si mismo con cierto grado de suficiencia, ya que la compilación es bastante problemática.

Existen cuatro métodos para compilar **Hurd**:

- **crossHurd de Debian. ***
- **GSC. GNU.**
- **build.sh ***
- **a mano.**

* No funciona actualmente, de forma correcta.

Los "sources" son un Tarball gigante. Incluye lo que podríamos llamar un sistema base. **Pancake**, esta preparando un script de instalación que funcione bien, pero todavía está en desarrollo. Es decir, que no funciona. La realidad, es que todo el mundo que tiene instalado el **Hurd** actualmente, lo ha compilado "a mano". Como ya se ha comentado, el proceso de compilación no es sencillo. Hay que ir probando hasta conseguirlo. Para ello, tendremos que proveernos de las herramientas apropiadas. Estas herramientas son:

- **gcc, gawk, binutils, sed**

¿Donde lograrlas?

- <http://ftp.gnu.org/>
- <http://bee.nepcode.org>

Nota: pancake ha desarrollado una distribución de **GnuHurd** que se llama **Bee**.

Como se ha indicado, la compilación es bastante complicada. Los pasos necesarios para completar la instalación son los siguientes:

- **create cross-binutils**
- **start making gcc cross-compiler**
- **Install Mach headers.**
- ...

Ahora vamos a ver una instalación desde binarios. La instalamos en una máquina virtual **qemu**. Esta acción es una precaución lógica. Gracias a las máquinas virtuales de **qemu**, podemos utilizar un **macintosh**, a pesar de que **Hurd** todavía no soporta esta arquitectura.

Lo primero que hay que hacer, es generar la imagen de un disco. Esto es, generar un disco duro entero en un solo fichero. Para ello, "Pancake" utiliza un script que ha realizado él mismo. Al utilizarlo, lo único que es preciso cambiar es el tamaño del disco. Las particiones las vamos a hacer mediante **fdisk**, aunque forzando un poco las cosas, ya que no estamos manipulando realmente un disco físico. Algunos datos hay que escribirlos a mano. Realizamos una partición. No merece la pena hacer más. Más adelante, formateamos la imagen.

Finalmente, montamos la imagen. En los **Unix** antiguos, había dos cintas, una para **/usr** y otra para el sistema base. De ahí viene lo de separar el sistema operativo de los ficheros de los usuarios.

Generar una imagen para **qemu** es un proceso bastante simple. configuramos el lector de cdrom y la partición de arranque. **HURD** Debería arrancar, aunque da errores la primera vez. En el segundo arranque, ya se habrán instalado los **translators** (traductores) y no encontraremos estos errores. Lo que nos encontramos tras arrancar, es la típica consola de texto. No hay nada más. La

filosofía de la distro **BEE** (obra de Pancake), es reducir al mínimo el sistema para lograr un arranque lo más limpio posible.

Pancake ejecuta **qemu**. Aparece el **grub** (gestor de arranque). **Bee Hurd**. El hardware es virtual. **Hurd** reconoce el hardware, los **translators**, etc. Hoy en día es un núcleo funcional, aunque da lugar a demasiados errores, como para poderlo utilizar en un entorno de producción.

Nota: Aparte de **HURD** existen otros micronúcleos interesantes como **Plan 9** o **Inferno**, pero no son tan ambiciosos como **Hurd**.

5 : 11 : 05.

- Charla Primera. Fundamentos Match y Hurd. Por Marchesi.

Un pequeño retraso. En lugar de comenzar a las 11 de la mañana, empezamos... **Marchesi** da un poco de tiempo a la gente que falta. Para mantener la atención del resto nos ofrece (como no) su ya célebre demo de **Emacs**.

Comenzamos. Utilizamos **Xfig**(programa de diagramas) como pizarra, aunque también se ha habilitado una pizarra física.

El **Hurd** corriendo en pareja con un micronúcleo conforma un núcleo. Un núcleo es un programa que sirve para salvar un nivel de abstracción. Toma una maquina (física) y presenta la información en un nivel de abstracción bastante más alto. Cuanto más se asciende (en el nivel de abstracción) más fácil es de comprender y utilizar. Por ello es tan útil abstraer niveles.

Los primeros núcleos fueron monolíticos. Estos, hacen algunas cosas feas aunque también poseen algunas ventajas. Las desventajas de los núcleos monolíticos son varias. Una de ellas, es la falta de seguridad. Otra, es la obsolescencia de los protocolos. Estos están basados en interfaces procedurales (puras convenciones). Por ello, Es bastante difícil mantener drivers al día. Cada vez que se hace una versión del núcleo, te ves obligado a adaptarte. La protección es otro problema, ya que todo está en un único bloque. Estas dos desventajas son problemas implícitos de la arquitectura de los núcleos monolíticos, por lo cual no se puede esperar soluciones. De haberlas, implicaría un rediseño a nivel de arquitectura básica (y de hacerse ya no estaríamos hablando de un núcleo monolítico).

Veamos un ejemplo acerca de esta problemática. Supongamos que tenemos una avería en un disco duro. Como consecuencia de ello todo el núcleo se viene abajo. ¿Es lógico que ocurra esto por un daño mínimo? Creemos que no, pero los núcleos monolíticos presentan este tipo de problemas.

Como alternativa a los núcleos monolíticos, se ha pensado en varias alternativas, entre las que destaca la aproximación conocida como "micronúcleo". Es conveniente saber que esta aproximación no da lugar a un único micronúcleo sino a varios,

cada uno de ellos con un enfoque diferente. **POSIX**, es un estandar para gestionar el acceso del núcleo a los procesos. **POSIX** es una maravilla. Es un programa muy sofisticado y capaz. Sin embargo, no es imprescindible que vaya vinculado a un núcleo monolítico, sino que hasta llegar a **Posix** podemos separar el gran núcleo, en unidades más manejables. Este enfoque, es el que utilizan los micronúcleos. El resultado, es un gran incremento de estabilidad y seguridad. El núcleo corre en modo "no supervisor", lo cual es una garantía de protección.

La comunicación entre procesos se realiza mediante "mensajes". Pasamos de una arquitectura basada en convenciones (interfaz procedural) a otra basada en mensajes.

Pasemos a otra enfoque: "el **exokernel**". No es una idea muy conocida. Sin embargo, es un concepto muy interesante. Lo ha desarrollado el **MIT**. Consiste en que el núcleo esté fuera. Utilizando una librería para comunicarlo con los procesos. **El exokernel** presenta un nivel de abstracción mínimo (y esto es ventajoso). Pero los librerías presentan un nivel de abstracción bastante mayor. Supuestamente, esta forma de hacer las cosas funciona. En algunas arquitecturas especiales (por ejemplo los sistemas de tiempo real) se ven desarrollos en esta dirección.

Durante toda la charla, cuando hablamos de procesos, nos referimos a procesos **POSIX**. Estos, son los mismos para un kernel monolítico o para un micronúcleo como **Mach**. Lo único que cambia, (cuando hacemos referencia a micronúcleos) es la necesidad de descender un nivel de abstracción. Ha habido cierta confusión sobre este tema. Crea dificultades el tema de los procesos **POSIX** y las tareas **Mach**. Las estrategias para generar los procesos dependen del sistema operativo que se utilice. **MacosX (Darwin)** lo hace de una forma y el **Hurd** lo hace de otra diferente. A partir de este momento, cuando nos referimos a núcleo, nos estamos refiriendo específicamente a **Mach**.

Hurd corre encima de **Mach**. Es una especie de pareja. Juntos conforman un núcleo completo. **Mach** es el núcleo y **Hurd** es una especie de capa de gestión que comunica **Mach** con **POSIX**. La filosofía fundamental de **Hurd** consiste en conceder mucha libertad al usuario, o mejor dicho el administrador del sistema. Entonces, ¿Qué es **Hurd** exactamente? **Hurd** es un grupo de tareas **Mach** que implementan **POSIX**. Así de sencillo.

Ahora vamos a pasar a explicar con detalle el núcleo **Mach**. Nos basamos (para la charla) en un documento llamado: "**Mach de Pe a Pa, obra de Marchesi**". Lo primero que vamos a ver son las abstracciones que implementa **Mach**. En primer lugar, hilos y tareas.

Una tarea es un container (contenedor) de procesos. Las tareas **Mach** contienen hilos de ejecución (procesos ligeros), y pedacitos de memoria (rangos de memoria). La tarea **Mach** no ejecuta nada por si misma, sino que es solamente un contenedor. Un proceso **POSIX**, por el contrario, si que ejecuta código (esto explica la diferencia entre una tarea **Mach** y un proceso **POSIX**).

Los hilos son unidades de ejecución. Se dice que son procesos ligeros. Hay gente que los considera una idea impresionante. Sin embargo no todo el mundo está de acuerdo con esta opinión (entre estos últimos "Marchesi"). Según este punto de vista, los hilos son en muchas ocasiones innecesarios y problemáticos. Las tareas **Mach**, se estructuran en forma de árbol. La unidad activa es el hilo. Son los hilos los que llevan los mensajes. Pero veamos ¿Que puede hacer un hilo? en pocas palabras: ejecutar código.

Un proceso **POSIX** está autorizado para hacer pocas cosas. El tipo de cosas que no se consideran peligrosas (sumar, restar, etc). Cuando un proceso **POSIX** quiere hacer algo más importante, está obligado a hacer una llamada al sistema. O sea, se lo pide al núcleo (que si está autorizado para trabajar con este tipo de cosas, para los que siempre son necesarios permisos). Las llamadas al sistema se suelen llamar "**Traps**".

En esta parte, "Marchesi" muestra los diferentes tipos de **Traps** que se utilizan. Dos hilos de la misma tarea se comunican mediante la memoria compartida. No es necesario avisar al núcleo para ello.

Un puerto es una vía de comunicación unidireccional. Los puertos son abstracciones independientes de las tareas. Supongamos que dos tareas se quieren comunicar. En **Mach**, se crea un puerto para permitirlo. Este puerto existe independientemente de la tarea. Es como un buzón. ¿Quién tiene los permisos de este buzón? Esto depende de la gestión de derechos de puertos. Un puerto, en un momento dado, solo puede ser revisado por una tarea. Sin embargo, otras tareas pueden enviar mensajes. ¿Entonces, como se gestionan los permisos? Es el núcleo el que gestiona las capacidades o derechos de los puertos. Hay tres tipos de capacidades o permisos. Las tareas como tales no gestionan puertos. Esto lo hace el núcleo. Las tareas solo gestionan unos índices. En el espacio de derechos de puertos de una tarea, la tarea solo envía un número. Un puerto es siempre unidireccional y posee una cola de mensajes dentro. El número (por decirlo de algún modo) de la tarea sirve para que el núcleo de permisos a la tarea para que esta pueda utilizar el puerto. El puerto no está asociado directamente a una tarea. El número es parecido a los descriptores en las tareas **POSIX**.

La comunicación de dos tareas entre ellas, implica enviar los derechos a través del puerto. A priori, las tareas no se ven entre ellas. Para comunicarse necesitan puertos. Y una tarea especial (similar a un listín telefónico) ofrece la información imprescindible para que las tareas puedan comunicarse. Esta parte de la conferencia da lugar a un intenso debate. Como las tareas no se ven entre sí, tampoco saben de su existencia. Para ello está el núcleo, y la tarea que hemos llamado "listín telefónico".

El hecho de que los puertos sean unidireccionales no impide las réplicas entre procesos. El truco consiste, en que cuando una tarea envía un mensaje a otra tarea a través de un puerto, también envía (embebido en el mensaje) los derechos de uso de los puertos. Estos permisos, viajan a través de los puertos y permiten las réplicas. De esta manera, y sin nada más, ya se pueden hacer muchas cosas. Hemos visto un ejemplo, que consiste en que una tarea solicita un par de bloques de disco a otra

tarea, que podría ser un gestor de sistemas de ficheros, y una tercera tarea que autentifique a la tarea inicial.

En este momento, tuvimos un nuevo debate. Hay un grupo de compañeros que desconfía de esta arquitectura. Se preguntan por el rendimiento de estos métodos, por su tolerancia a fallos, y por su nivel de seguridad ante ataques maliciosos. Pese a todo, parece ser que esta arquitectura es más flexible que la de los núcleos monolíticos. Si un núcleo monolítico peta, se acabó. Si un micronúcleo peta, pues también se acabó. Sin embargo, en el mundo de los micronúcleos es posible diseñar tareas que gestionen o controlen a otras tareas, lo cual puede servir para tener un control más preciso de este tipo de situaciones. Los milagros no existen. Una tarea que ha petado no va a relanzarse sola. Si así lo hicieran, no estaríamos aquí ahora. Las máquinas ya gobernarían el mundo, por si mismas...

Bien, ¿Como se crea una tarea nueva? Ya sabemos que las crea el núcleo (**Mach**) pero claro ¿Quien le hace la petición? Esta claro: otro hilo de otra tarea. No hay otra posibilidad. ¿Cual es la primera tarea de todas? Esto se vera más adelante.

Los procesos en **Mach** tienen jerarquías, al igual que ocurre con los procesos **POSIX**. Una tarea siempre tiene un proceso padre. Sin embargo, no importa si la tarea padre ya no está viva (en **Mach**). En **POSIX** es diferente. No puede haber procesos en marcha si su proceso padre ha terminado.

Las tareas tienen espacios de capacidades de comunicación (permisos), y un rango de memoria. La tarea hija no hereda los permisos (carecería de sentido, ya que el hijo seria un clon del padre). El listín telefónico, es una tarea (lo llamamos así porque queremos). Pasemos ahora a los puertos del nucleo. Las tareas contienen un array para los puertos registrados. "Marchesi" considera esto un contrasentido. En lugar de hacer las cosas bien, es decir quitar el soporte de puertos registrados, se ha hecho este array. Suponemos que se ha incluido por cuestiones de eficiencia, pero se carga la filosofía del diseño de la arquitectura de **Mach**. Este array es claramente una chapuza.

Para suspender una tarea se hace una llamada al núcleo y se identifica la tarea mediante su puerto núcleo.

Las tareas envían mensajes. Pero no sabemos nada de los mensajes en sí. Los mensajes en **Mach** son elementos muy flexibles. Supongamos que queremos enviar una cantidad enorme de memoria de un proceso a otro. Primera consideración ¿Donde tiene el proceso lo que quiere enviar? Respuesta: en su memoria virtual. Segunda consideración ¿Como se envía? Respuesta: como siempre, mediante un mensaje. La idea de **Mach** es enviar los datos off line. (las iniciales: **COW**, significan copy on write). Dicho de otro modo: en un mensaje se puede enviar cualquier tipo de información.

En un micronúcleo, un gestor de memoria se puede correr en modo no supervisor. Esto es bastante asombroso. El núcleo **Mach** tiene un paginador que gestiona solamente la memoria física.

Conceptualmente, ayuda bastante ver al propio micronúcleo como

si fuera una tarea en modo supervisor (de hecho es una tarea).

**- Charla Segunda. M. Virtual en Mach. Paginadores en Hurd
Por Koro.**

Esta charla es conceptual. No se va a ver código. ¿Por que? Porque **GNU** autodocumenta todos sus proyectos. Lo primero que vamos a ver son las peculiaridades del sistema **Mach/Hurd**.

El elemento base se llama "objeto de memoria". Los paginadores se colocan fuera del núcleo. La finalidad de los micronúcleos consiste en estar fuera y gestionar, con la máxima flexibilidad posible, todos los servicios necesarios.

Veamos ahora lo que son los objetos de memoria a los que acabamos de hacer referencia. Un objeto de memoria es una estructura que define una región de memoria. Reside en el núcleo. Almacena referencias a las páginas que forman una región de la memoria. **Gnu/Hurd** no hace reservas de memoria de verdad. Acondiciona la memoria limpiándola (rellenando con ceros). Es necesario reconocer que esta forma de proceder no aporta en principio ningún beneficio.

Un paginador es considerado una tarea más. Una tarea en espacio de usuario. Se considera que es un servidor en si mismo. Un objeto de memoria podría compararse con un sistema de archivos.

6 : 11 : 05.

- Charla Primera. . Mach IPC y MIG RPC. Por Marchesi

IPC (interprocess communication). Utilizando este sistema, las tareas pueden comunicarse entre sí. Tres abstracciones permiten dicha comunicación:

- Estructuras que forman los mensajes (`mach_msg_header_t`)
- Estructuras que identifican puertos (`mach_port_t`)
- La primitiva de `env`{i}`. La rutina `mach_msg`.

NOTA. ¿Cuántos mensajes se envían en un ejemplo que consistiría en multiplicar dos números? Dos. ¿Cuántos puertos están involucrados? dos también (recordemos que son unidireccionales).

Los diálogos entre tareas son un tema más complejo de lo que pueda parecer. La idea de **RPC** y **CORBA** consiste en transformar las operaciones entre procesos en funciones. Una tarea envía un mensaje, otra lo recibe. Opera y devuelve un valor. La primera tarea espera. Es igual que una función.

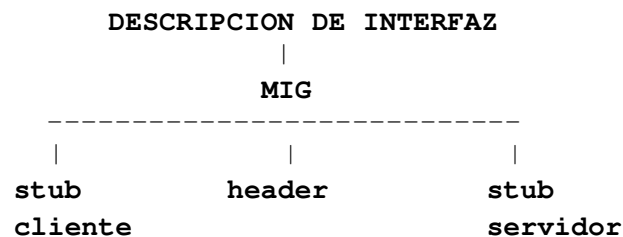
La comunicación entre tareas en **Mach** es muy flexible. Esto es bastante bueno. Sin embargo, da lugar a una gran complejidad de la interfaz. **IPC**, no es ni más ni menos que el sistema que permite (en la práctica) la comunicación entre tareas. Para lograr esto, hay que hacer muchas otras cosas. Hemos visto un ejemplo con anterioridad. En el otro lado (tarea que espera la petición) tiene lugar el mismo proceso.

Antes, tuvimos un debate sobre la complejidad del sistema de tareas en **Mach**. Vimos como enviar una tarea. Bien. Si no se hubiese buscado ninguna alternativa, todo tendría que hacerse de este modo. Lo que se ha hecho es utilizar **RPC** (Remote Procedure Call). **RPC** simplifica mucho nuestro trabajo. Este enfoque requiere un generador de interfaces para funcionar. El generador de interfaces se llama **MIG**.

Hemos visto un ejemplo con dos tareas **mach** y una función llamada **sumadosnum**, que sirve para sumar dos números. La gracia del asunto está en que nosotros solo escribimos la función. Las estructuras para los mensajes se generan automáticamente. O mejor dicho, lo hace **IPC** automáticamente. Si se desea, se puede utilizar **IPC** picando el código a mano (a veces puede ser útil).

Los sistemas construidos sobre **Mach** tienden a utilizar un modelo cliente-servidor. Vemos un ejemplo con dos tareas. Una hace operaciones matemáticas, y la otra pinta en la pantalla. Las dos tareas son cliente y servidor.

El **HURD**, utiliza también un modelo cliente-servidor. Veamos ahora como opera **MIG**:



MIG es un programa como otro cualquiera. Parte de un solo fichero y genera tres, como puede verse en el gráfico anterior.

Para intentar entender como funciona todo esto, vamos a ver un ejemplo:

clock.defs --> es un fichero de definición de interfaz (en este caso un reloj). Estas interfaces pueden ser consideradas también protocolos de comunicaciones. Dime lo que haces y te diré quien eres. Supongamos que estamos desarrollando para **Hurd**, y queremos hacer un paginador externo. Las interfaces definen protocolos.

Nota >Esto ayuda a entender las fuentes del **HURD**, porque tienen mucho de esta lógica.

Clock es un protocolo-interfaz de servicios horarios. Podremos implementarlo en cualquier tarea donde nos resulte útil. Supongamos que hacemos una tarea concreta (hora), cuyo trabajo consiste en tomar la hora y modificarla. Escribimos el cliente, tomamos la **interface clock** y deberíamos hacer un servidor. Este tendría que ser compilado por **gcc** para obtener un binario. Si se quiere que una tarea hable chino, se enlaza con el **stub** que

implementa el chino. En definitiva, tenemos dos programas, el servidor y el cliente, escritos ambos en **C**.

mach_msg_server (rutina de servidor de envío de mensajes). Si no la utilizamos, tendríamos que picar nosotros a mano cada rutina.

Si se quieren utilizar varios protocolos, son necesarias rutinas multiplexadas (y mensajes multiplexados). Son imprescindibles. Un compañero sugiere otra solución: Darle a cada protocolo un puerto. "Marchesi" supone que funcionaria. Las estrategias en **Mach** son bastante abiertas. Estas estrategias son un poco las que a cada cual le funcionen bien.

La idea de todo esto es que (gracias a **MIG**) el desarrollador se pueda abstraer de las cuestiones relativas a puertos, mensajes, etc y concentrarse en su trabajo real: "Implementar los servicios utilizando las rutinas de servicios". El código resultante es bastante pequeño. Esto es ventajoso.

Un protocolo puede tener definido varias interfaces, aunque suene un poco raro. La mensajería en **Mach** es asíncrona. Cualquier invocación a una función es síncrona. Esto es un poco contradictorio, pero es útil en algunas situaciones. Si una tarea hace un **RPC** asíncrono, debe de tener un puerto que envíe la respuesta. Cuando hacemos una notificación a otra tarea, continuando sin esperar respuesta, es cuando esta técnica demuestra su utilidad. Pero en general, no es demasiado útil una **RPC** asíncrona.

NOTA >Es un ejercicio muy interesante leer lo que produce **MIG**. También por supuesto, las fuentes.

- Charla Segunda. Implementacion en GnuHurd. Por Koro.

La mayoría de las interfaces en **Mach** se almacenan como librerías. Estamos viendo como el servidor implementa la interfaz. Ahora vamos a verlo en la tarea cliente. Esta tarea se implementa con **Glibc**. En todo el **Hurd**. El cliente está en **Glibc** como norma general, por compatibilidad con **POSIX**. Los **stub**, solo aparecen después de la compilación. Si los buscamos en las fuentes no los vamos a encontrar. Hay que compilar **Hurd** y **Glibc**.

Pasamos a ver el arbol de directorios de **Hurd**. Cuando se inicializa una tarea (servidor) y quieres que te localicen, vinculas una tarea a un puerto. Como listín telefónico se utiliza el sistema de archivos. Ahora arrancamos un **Hurd** mediante un **Qemu** en el PowerPc de Jemarch. Después montamos el lector de CD-Rom (virtual, claro. Estamos en **Qemu**):

```
settrans -a /cdrom /hurd/iso9660fs /dev/hd1
```

-a es para que el servidor sea activo.

Un nodo puede ser un fichero cualquiera. Con **touch** hacemos un

fichero, de nombre hello. Sobre hello montamos el traductor hello. Un traductor es una tarea. Los translators son servidores que implementan servicios. Una tarea que no es un traductor (translator) es **ls**, por ejemplo. Para simplificar, la mayor parte de los servicios (translator) se implementan como librerías.

la cabecera es el nombre de la librería **-lib**, y **.C**, por supuesto. Lo más interesante es empezar estudiando las librerías.

```
ext2(servidor)>
/home/ ext2(servidor)>
jemarch/ hello (servidor)
hello
```

- Charla Tercera. Micronucleos. Por Koro.

Existen otros micronúcleos aparte de **Mach**. El más conocido es **L4** que es un proyecto desarrollado y mantenido por una sola persona. Su enfoque consiste en hacer un microkernel lo más pequeño y limpio posible. **L4** gusta a mucha gente, aunque también tiene detractores.

Nota> El **Minix 1.0** es un núcleo que se puede estudiar en una semana. Está muy bien verlo. Minix no tiene soporte para memoria intermedia. Si una tarea envía un mensaje a otra tarea, y esta última no está preparada para recibirlo, el sistema peta inevitablemente.

Los hilos migratorios vienen de la teoría de objetos pasivos. **L4** no tiene unas políticas amplias de gestión de memoria (entre otras muchas cosas) De algún modo, se puede decir que **L4** es un núcleo recortado. La magia no existe, si algo no lo hace el núcleo, alguien tendrá que hacerlo. ¿Quién? el usuario, naturalmente. La IPC de **Mach** es rica, la de **L4** es pobre.

¿Quiere esto decir que **L4** es un mal proyecto? No. Es un proyecto interesante, pero para poder hacerlo funcionar bien habría que implementar todo lo que le falte en entorno de usuario.

Teóricamente, **Windows NT**, era en origen también un micronúcleo. Pero con el paso del tiempo, ya no se puede decir que todavía sea un micronúcleo. Hay que añadir, que no sabemos realmente como es el núcleo de **Windows NT** (es privativo). Tenemos que creernos lo que nos cuente **Microsoft**. Y la verdad, es que no nos lo que creemos. El caso de **Darwin** es bastante peculiar. Resulta que **Darwin** tiene un rendimiento pobre (comparado con **Gnu/Linux**). Entonces ¿Porque **Aqua** tiene un buen rendimiento. No lo sabemos. Es propietario. Suponemos que hay algún truco, relacionado con procesos en tiempo real.

Otro microkernel interesante es **Eros**. Este proyecto tiene como objetivo realizar el núcleo más seguro que haya existido jamás.

ANEXO1. RECURSOS WEB RELACIONADOS CON GNUHURD.

- irc.freenode.net
- hurd-es
- http://es.gnu.org/hackerscentral
- bee.es.gnu.org
- es.gnu.org/cgi-bin/hurd.pl

Antonio Becerro Martinez

littleddog@es.gnu.org

Madrid - 2005

Se permite la copia del artículo completo en cualquier formato, ya sea sin ánimo de lucro o con fines comerciales, siempre y cuando no se modifique su contenido, se respete su autoría y esta nota se mantenga.

